JPA flow-1

In this flow, we introduce Object Relational Mapping with JPA. All designs will be "simple" with no relations between classes (and therefore tables), this will come in flow 2.

Before you start with these exercises you should have completed the guide [JPA with NetBeans (my first java-JPA application)]() to learn how to use JPA with NetBeans and our recommended developer setup

EX-1 JPA, Entity Classes, transactions and cool JPA-Queries.

This exercise draws on a tutorial from [objectdb.com](). Objectdb is a full ObjectOriented database that implements the JPA interface. We will not use the objectdb-database, but eclipselink as our implementation, but they provide some of the best documentation for JPA, so we will use their documentation in several places.

1) First, create a new database for this project called **point** on your docker MySQL instance (remember `docker-compose up -d`)

2) With NetBeans, create a new plain Maven Web Java Project called **point** and add an Entity class **Point** to the project (in a package `entity)`. Do this similar to how you did it in the "*... my first java-JPA application*" and while you do it, make sure to create a persistence.xml file that points to your point-database.

- Remember to add the mysql-connector to the pom-file.
- Add this dependency to the pom-file
- Open the persistence.xml file and change the provided `Persistence Unit Name` into **pu**

3) In the Point entity class, delete the generated `hashcode(), equals()` and `toString()` methods (you can always create them again by right-clicking and selecting insert Code…

Change the `@GeneratedValue` annotation to use the **identity** strategy, and understand why this is the right thing to do, given that we use MySQL.

4)

- Add two int fields, x and y, to the class to represent a point
- Add getters and setters for the two fields (again right-click and select insert Code…)
- Create a constructor that takes an x and y value (remember the necessary constructor)
- Add a new toString() method to the class.

5) Add a Main class to the project.

Create a plain java class called **Main**, and copy all the content given below into this class.

```
import javax.persistence.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // Open a database connection
        // (create a new database if it doesn't exist yet):
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu");
        EntityManager em = emf.createEntityManager();

        // Store 1000 Point objects in the database:
        em.getTransaction().begin();
        for (int i = 0; i < 1000; i++) {
            Point p = new Point(i, i);
            em.persist(p);
        }
        em.getTransaction().commit();

        // Find the number of Point objects in the database:
        Query q1 = em.createQuery("SELECT COUNT(p) FROM Point p");
        System.out.println("Total Points: " + q1.getSingleResult());

        // Find the average X value:
        Query q2 = em.createQuery("SELECT AVG(p.x) FROM Point p");
        System.out.println("Average X: " + q2.getSingleResult());

        // Retrieve all the Point objects from the database:
        TypedQuery<Point> query = em.createQuery("SELECT p FROM Point p", Point.class);
        List<Point> results = query.getResultList();
        for (Point p : results) {
            System.out.println(p);
        }

        // Close the database connection:
        em.close();
        emf.close();
    }
}
```

6) Build the project

If you get this error: `Failed to execute goal`

```
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile (default-compile) on project

point: Fatal error compiling: java.lang.NoClassDefFoundError: javax/annotation/Generated:

javax.annotation.Generated
```

Add this dependency to the Pom-file (or downgrade your SDK to version 1.8)

```
<dependency>
            <groupId>javax.annotation</groupId>
            <artifactId>javax.annotation-api</artifactId>
            <version>1.3.2</version>
</dependency>
```

7) Finally, run the code
Run the Main class, and if everything was fine, use Workbench to verify that a POINT table was created and populated with 1000 rows.

8) Before you continue
Make sure to read the code in the Main-class. It provides several examples you can use as inspiration for future exercises as for example:

- How to use entity transactions
- How to get the number of rows in a table
- How to get the average of a numeric value
- How to get all rows in a table

Important: There is NO SQL in the Main-class, what you see is **JPQL** - understand the difference!

Ex-2 The Entity Class and Facades

Create a new **database** for this exercise

With NetBeans, create a new plain Maven Java Project and use the Wizard to add an Entity class **`Customer`** to the project (in a package called **`entity`**)

Initially add the following fields: **`firstName`** (String), **`lastName`** (String) and **`created`** (java.util.Date) and provide getters and setters for the two name-fields, and (only) a getter for the `created` field.

Hint: Dates require a special JPA.annotation. Use the link we have provided for all JPA annotations and read about `Temporal,` and then use it.

Provide the class with a constructor that takes the two name values, and sets the `created` field to the current time (important, remember the constructor you always need)

Provide the entity package with a class `EntityTested.java` with a `public static main(..) method` and Instantiate and persist a few Customers, similar to what you did in the getting started guide.

Ex-2 Continued - Adding a facade

Add a facade to the project, using the template given in the section *JPA With a Facade* in the intro tutorial.
Provide the facade with the following methods:

```
Customer findByID(int id);
List<Customer> findByLastName(String name);
int getNumberOfCustomers();
List<Customer> allCustomers();
Customer addCustomer(String fName, String lName);
```

Test the facade, first, with a simple main method, similar to what was done in the intro tutorial. Next step, however, will require you to write a JUnit test for the facade.

🟨 🟥 Ex-3 Add a JUnit Test to your facade

We will have much more focus on test in week-3, so take this as an appetizer to how we are going to develop throughout the semester. Today, this is marked as yellow/red, but later you are all expected to provide automated tests for your code.

For this exercise just use the database used up until now. When we start to do testing “for real” we will add an additional database, meant only for testing.

Create a new JUnit test for your facade class, using the wizard “`Test for Existing Class`”

Use your knowledge from the previous semester and try to design a test that will ensure that the database is in a well-known state before EACH test.

Test the facade methods in this test.